**Box 2D**

Box2D**1** is an open source physics engine, specializing in simulating rigid bodies in 2-dimensions. It was originally developed by Erin Catto in C++, but has since been expanded to multiple platforms and languages. Box2D utilizes many features and algorithms that will be useful in developing a physics engine. For instance, Box2D simplifies differential equations (the form taken by many equations when doing physical calculations) by using numerical integration to solve approximately.

This numerical integration greatly reduces calculation time (thus speeding up the game loop) without sacrificing too much precision (at least unnoticeable by humans during gameplay). For simple Newtonian physics problems solving is usually easy (ie the distance a baseball will travel, simply solve d^2y/dt^2 = -g for the time when the ball reaches y=0, and calculate the distance based on the initial velocity.) However, solving complicated problems introduces hard to solve differential equations (ie the distance a baseball will travel taking into account wind resistance). These non-linear and multi-variable problems are the best candidates for approximation via numerical integration.

Numerical integration is done by starting with the first order form of our equation. Using the baseball example, we have our velocity equation dx/dt = f(t, x) and our initial velocity x(0) = x0. Assuming f(t,x) is some non-linear function, we can approximate the slope (ie velocity dx/dt) with a linear function [x(t + h) – x(t)] / h. Rearranging terms we have the approximation:

X(t + h) = x(t) + h f(t, x(t))

This formula allows us to get a reasonable approximation for some time step h, and get the position x at any time, with a much smaller CPU load. Of course the larger h is, the larger our uncertainties are, so its best to keep h small by recalculating x(t+h) every game loop.

Constraints are also a aspect to keep in mind for physics engines, something that Box2d takes care of well. The constraint matrix **C** is given by **C=Jv** + b, as shown through classical mechanics. We can use the geometry to write the constraints **C,** differentiate C with respect to time, isolate v, and identify J and b by inspection. Typical constraints to take into account include joints, motors, contact, restitution and friction.

Of course, there is much more to these subjects and much more needed to completely cover all the physics required to fully describe rigid body dynamics, but Box2D’s algorithms for constraints and simplifying differential equations is a good start.

**Interactive Simulation of Rigid Body Dynamics in Computer Graphics**

The research paper written by Jan Bender, Kenny Erleben, Jeff Tinkle, and Erwin Coumans, entitled **Interactive Simulation of Rigid Body Dynamics in Computer Graphics3** gives a very good overview of simulation techniques in regards to rigid bodies. Some key basics for physics engines to keep in mind that the paper brings up includes the following ideas.

Simulation speed is very important. Most physics engines must be able to operate at or above 60 FPS (frames per second) under normal, expected loads. Obviously, if the physics engine is operating below this optimum level, it will create bottlenecks in the game loop, creating lag and ultimately a bad experience for players. Of course doing extremely accurate physical computations will definitely lead to bottlenecks, so many physics engines (especially those geared towards simulating physics for games) will need to make sacrifices in accuracy to adhere to the 60 FPS rule. Techniques such as numerical integration and approximating non-linear functions to the first order are common ways to do this (see above).

Bottlenecks can also be deterred in the structure of the game loop. In video games, a game loop generally refers to the central loop that performs various calculations every cycle that will be needed for the game (such as adjusting the score, or moving particles on the screen, or playing sound). The physics engine will sit inside the game loop and perform various calculations that are also needed. The traditional simulation loop, as described by Bender, consists of Collision Detection -> Contact Handling and Collision Resolution -> Time Integration -> Repeat.

However, this simplified simulation loop has areas for improvement that will improve overall FPS. As Bender suggests, we can avoid bottlenecks by simplifying the collision detection phase into 3 simple phases, broad phase, mid phase and narrow phase. In the broad phase, bodies will be approximated by simple geometries (usually a circle that completely covers the object). We can then broadly check the circles for overlaps. Since each circle fully covers an object, if two object’s circles do not overlap, neither will the objects. This quick calculation allows us to completely ignore these objects for the collision detection phase.

In the mid phase, we take a closer look at objects whose circles had overlapped in broad phase. We cull parts in the local body space, culling by using bounding volume hierarchies. Once we have these, we enter the narrow phase, where detailed geometries of the bodies are used to find precise body features in contact and locations of contact points. Of course not all games will require such precision as contact points since most games depend solely on contact, some games will require this precision (such as shooters, if you shoot a person in the foot vs head, etc).

It is important to balance accuracy and balance in physics engines, and as Bender points out, these will constantly be in battle with a physics engine. The goal of the engine is not necessarily to provide the game with the **exact** outcomes of a interaction as it would in real-life, but to provide something that is believable.

One example of this simplification is in the idea of rigid bodies. A rigid body by definition is an idealized solid object for which the distance between every pair of points on the object will never change, even under the stress of huge forces. Of course this is not realistic as objects will compress, however so slightly under forces (even something as solid as a diamond will compress ever so slightly under even small forces). We can greatly reduce the complexity and calculation costs by assuming that these negligible compressions can be ignored by considering “hard” objects as completely rigid.

**An Introduction to Physically Based Modeling**

David Baraff, of Carnegie Mellon University, writes a similar paper in **An Introduction to Physically Based Modeling2** where he writes about the basics of unconstrained rigid body dynamics. Baraff first discusses the basics of classical mechanics, including differential equations and kinematic equations. This leads us to the rigid body equation of motion, defined by the matrix Y(t) = {x(t), R(t), P(t), L(t)} which will define the state of our rigid body. In this equation, x(t) is the position, R(t) rotation matrix, P(t) linear momentum and L(t) angular momentum. These equations will fully describe our rigid body as will be necessary for a physics engine to calculate. Since these equations are well defined and proven physical descriptions, I will not prove them again here.

Baraff shows acceptable ways to calculate and solve for Y(t) via computer algorithms in his paper in C programming language. Using structs and arrays to store the matrixes and values to the equations listed above. Functions of course will be called during the game loop to update each object’s Y(t) and the game loop will query each object to determine its state and make the appropriate movements. Baraff goes on to describe fundamentals of classical mechanics and Newtonian physics.

**Asynchronous tasks in Android**

Java and Android has some built in features to assist developers in using asynchronous tasks (ie parallel computing)4. This is done by having an object implement *Runnable* and an object that is a subclass of *Thread*.

The object implementing Runnable will need to override run() with the tasks that need to be done (ie check for a collision). The Runnable object will eventually be passed to the Thread constructor, and all the contents of run() will be performed in a separate thread.

The thread class allows for concurrent execution of runnable objects. Thread classes will have its own separate call stack for all methods, arguments and variables. In essence, each thread is characterized as its own virtual machine running code independently of other threads.

To interact between threads, objects can be shared (so two threads can read and/or write to the same memory address/object). Synchronized methods will also allow threads to cooperate. To initiate a thread one a runnable object is given, the thread’s start() method will need to be invoked. Priorities of threads can also be changed via setPriority(int), which will determine which thread will get SPU priority in the case of limited CPUs.

Android essentially uses mostly pure Java to deal with threads and there isn’t a whole lot different from pure Java threads to Android threads. AndroidOS will take care of threads automatically, and determine (if on a multi-core phone of course) which threads should be run on separate CPUs and when. This determination is made mostly on CPU load, and if two threads are taking up a significant portion of CPU, AndroidOS will most likely separate these two threads. There is no way to specify which thread runs on which CPU.

Android also has native threads of its own that should not be interfered with (such as the UI thread, taking care of all video output etc). While it’s important to keep synchronized with the default android threads, it shouldn’t be a problem in this application, as only calculations solely for physics engine will be carried out in our private threads, and all other android necessary work will be done in the default threads as a normal android application would do.

**Parallelizing the Open Dynamics Engine**

Michelle Goodstein, Michael Ashley-Rollman, and Paul Zagieboylo undertook a similar task in making an existing physics engine into a parallelized physics engine, in their report **Parallelizing the Open Dynamics Engine**5. In this project, the authors used an existing open source engine (ODE) that is used for simulating rigid body dynamics in 3 dimensions. Being open source, the authors had access to all the source code so their task was solely to make it work in a parallel environment.

As the authors point out at the time of the writing, and searching through current physics engines I believe this is still true today, PhysX by Nvidia is the only general purpose physics engine that currently utilizes parallel algorithms. Though I couldn’t find any examples, I’m certain that very specific physics simulations exist in parallel, but from my search PhysX is the only true physics engine in the sense that it is versatile to a wide range of problems that is rigid body dynamics.

The authors spend a lot of time talking about specific functions and changes they made to ODE (written in C/C++), which I will not recount as it does not relate to my project, however the portions of code that they were able to parallelize with efficiency are pertinent. The best places they found to parallelize in an engine were the collision detection and solving linear equations, both of which I have previously described above.

Collision detection is a perfect candidate for parallel optimization because there are many comparisons needing to be made, most of which can be made in parallel because most comparisons are independent events (ie checking if A intersects with B and if C intersect with D are completely unrelated, and could be performed simultaneously in different threads). The authors tackled collision detection by allocating *n* threads, which collected collision information simultaneously on all objects and waited until all threads finished. Once all finished the main thread was signaled that all data was ready to be worked on and this process would repeat every game cycle as a normal physics engine would behave. Note data structure choice here is important, as something like a linked list would work great in single-thread physics engine will be very hard to parallelize.

Next solving systems of equations can also be parallelized. For instance, we can be solving for the state of object A while simultaneously solving for state of object B so long as we have all the forces that are being acted on them (which would be determined in the previous step). So long as all threads meet back up before going to the next step of applying the forces, we can calculate all information simultaneously on our *n* threads and again signal main thread when we are ready to proceed.

These two problems are best for parallelization because there is a lot of room to speed things up and there is lots of independent objects – the perfect case for parallelization. Experiments that the authors made against the original ODE did show significant improvements (note these numbers are just with the collision detection speedup, solving systems of equations was not implemented in final project due to time constraints). Collision detection accounted for 40% of CPU utilization in a world with 8000 tennis balls in original ODE, with 2 threads they had a 123% speed up, and in four threads a 226% speedup of the collision detection calculations.

While these numbers were quite good, these numbers can only be achieved with their algorithm in highly populated spaces. Smaller worlds for example, with only 1000 objects only had a 107% speedup with four threads, and actually performed worse than the original ODE when using less than four threads. This is of course because at only 1000 tennis balls, collision detection is not the bottle-neck, so optimizing collision detection at this point is pointless.

**Massively Parallel Rigid Multi-Body Dynamics**

Similar to the ODE paper above, Klaus Iglberger and Ulrich Rude also work on parallelizing a physics engine entitiled **Massively Parallel Rigid Multi-Body Dynamics** which take the ODE idea above to a massive scale6. Instead of altering an existing project like above, Iglberger develops his own physics engine (called *pe*) to be used for massive parallelization of rigid bodies. pe is a framework written in C++ and has been under development since 2006.

As Ilgberger points out, current implementations of physics engines come in two limitations. This includes engines that strive for accuracy but are polynomial time and thus will require massive computing power with additional objects and are limited to a few thousands concurrent objects. The other variety used mostly for games forgoes accuracy for speed, but these are also limited to a few thousand concurrent objects due to time restraints in games. They attempt to solve this problem by parallelizing a physics engine through MPI (Message Passing Interface).

Their algorithm first will assign each rigid body to a specific MPI. Because rigid bodies are not point masses, it is important to keep the entire rigid body on the same MPI the entire time to avoid discrepencies, since they partition their processes in space (a rigid body could be on the boundary and scale two processes). To avoid this the center of mass was used as a pin point on the location of the body. Each process checks the center of mass, and if it resides insides that process’s bounds, then it will be responsible for carrying out actions on that body. Each process has no knowledge of any other objects that are outside its boundary. If a body is overlapping two processes, both processes must work with that body because there could be forces acting on that body from both processes.

MPI processes send data to each other to stay in-sync. Obviously for performance and complexity reasons, the number of MPI messages sent and received should be kept to a minimum, which can be problematic for a large scale rigid body simulation. Therefore, for efficient parallelization we need to send a single MPI message to a remote process in every communication step. Since this requires multiple data structures to be sent in each MPI message, they solve this problem by sending raw bytes and having each process reconstruct the structures from the byte stream. Since a process cannot know beforehand if an incoming message is expected, even if no data is needed to be sent, they still need to send an empty message as the process must be waiting every step for this to work properly.

The parallel algorithm is based off of the “fast frictional dynamics” solver that was presented in a previous paper. This FFD solver is a fast real time collision response algorithm. The benefit of this FFD algorithm is the fact that collisions are treated locally – that is post-collision velocities for colliding rigid bodies are only calculated based off the states of the colliding rigid bodies. This means that no LCPs (linear complementarity problems) that many physics engines would need to solve. The FFD runs in O(n + c) time, where n is the number of rigid bodies and c is the contact points between bodies.

FFD consists of three main steps. Step 1 (after a MPI force synchronization) has every rigid body take a half step of it’s velocity. A simple for loop of all rigid bodies that belong to each Parallel process is ran to do this step.

Step 2 (after a MPI update of remote and notification of new rigid bodies) first checks all contact points for all rigid bodies. Another for loop for each process checks all rigid bodies that belong to said process for contact points. If a contact point is found, rotation and constraints are calculated.

Step 3 ( after a MPI exchanging constraints on rigid bodies) takes a further look at constraints violated in step 2. For all rigid bodies that had violations, post-collision velocity is calculated, friction response is taken into account and the resulting new velocity is added to the rigid body position. If no constraints were violated, then the original velocity is simply added again in another half-step to complete the full-step velocity. Finally another MPI is sent updating remote and notification of new rigid bodies before the process repeats at the force sync and step 1.

Of course one forseen problem is a collision taking place on a boundary. Since each process only knows of rigid bodies who have their center of mass within the process’s boundaries, its possible that the a collision is taking place without the local process knowing about it (consider a ball, whose center of mass is within the bounds, but the outer radius has a contact point outside of the bounds with a rigid body that is fully contained in the remote process bounds.) The remote process will see both rigid bodies, since the ball has points within its bounds, so the remote process must make the constraint calculations. But since the remote process does not control the ball, the constraints will then be sent via MPI to the local process for step 3.

Additionally, when a rigid body leaves the boundaries of the local process, the local process must inform the remote process that it crossed into of the event. This works as such: When the rigid body is fully contained in the local process, the remote knows nothing about it. When the overlap begins (but the center of mass and therefore the ownership still belongs to the local) the local process will send an MPI to the remote of its state (but ownership remains on local). When the center of mass cross the bounds then the remote process will take ownership of the rigid body (but local will still retain knowledge of its state). Finally, when the rigid body fully exists the local bounds, the local will cease knowing about the rigid body.

This was tested on the HLRB-II supercomputer, a computer consisting of 9,728 dual core processors which could easily simulate 1.4 billion rigid bodies, something completely undoable with a single core processor. Obviously for mobile development consisting of 2-4 cores, this is not comparable, but the basic algorithms can still be studied for similar approaches.

**My Implementation and Problems Encountered**

**Works Cited**

1. "Box2D." *Box2D*. N.p., n.d. Web. 07 Mar. 2013. <http://box2d.org/>.
2. Baraff, David. *An Introduction to Physically Based Modeling*. N.p.: n.p., n.d. Print.
3. Bender, Jan, Kenny Erleben, Jeff Tinkle, and Erwin Coumans. "Interactive Simulation of Rigid Body Dynamics in Computer Graphics." (2012): n. pag. Web. 6 Mar. 2013. <http://www.cs.rpi.edu/twiki/pub/RoboticsWeb/LabPublications/BETCstar\_part1.pdf>.
4. "AsyncTask." *Android Developers*. N.p., n.d. Web. 11 Mar. 2013. <http://developer.android.com/reference/android/os/AsyncTask.html>.
5. Goodstein, Michelle, Michael Ashley-Rollman, and Paul Zagieboylo. "Parallelizing the Open Dynamics Engine." CMU, n.d. Web. 07 Mar. 2013. <http://www.cs.cmu.edu/~mpa/ode/final\_report.html>.
6. Iglberger, Klaus, and Ulrich Rude. "Massively Parallel Rigid Multi-Body Dynamics." (2009): n. pag. University of Erlangen. Web. 6 Mar. 2013. <http://www10.informatik.uni-erlangen.de/Publications/TechnicalReports/TechRep09-8.pdf>.